Efficient SAT-Based Mapping and Scheduling of Homogeneous Synchronous Dataflow Graphs for Throughput Optimization

Weichen Liu[1], Mingxuan Yuan[1], Xiuqiang He[1], Zonghua Gu[1], Xue Liu[2]
[1]Dept of Computer Science and Engineering
Hong Kong Univ. of Science and Technology, China
[2]School of Computer Science
McGill University, Canada

Abstract

As Moore's law comes to an end, multiprocessor systems are becoming ubiquitous in today's embedded systems design. In this paper, we address the problem of mapping a Homogeneous Synchronous Dataflow (HSDF) graph onto a multiprocessor platform with the objective of maximizing system throughput. We present two optimization approaches based on branch-and-bound and SAT-solving to explore the design space of all possible actor-to-processor mappings and static order schedules on each processor. In the Logic-Based Benders Decomposition (LBBD) approach, we decompose the problem into a master problem of finding a feasible actor mapping and scheduling, and a sub-problem of deadlock-checking and throughput computation. In the Integrated approach, we integrate branch-and-bound search into the SAT engine to achieve more effective search tree pruning and better scalability. Performance evaluation shows that the Integrated approach outperforms the LBBD approach by a large margin.

1 Introduction

In the dataflow paradigm, a program is represented as a directed graph, where the nodes, called *actors*, represent computational modules, and the directed edges represent communication channels between the modules. *Synchronous Dataflow* (SDF) is a special type of dataflow model where each actor invocation (also called a *firing*) consumes and produces a constant number of data tokens. It is widely used in a broad class of signal processing and digital communications applications, including modems, multi-rate filter banks, and satellite receiver systems.

Given a SDF graph and a multiprocessor hardware platform, the design problem includes *mapping* (assigning) actors to processors, and *scheduling* the actor firings on each processor, so that all data dependency constraints are satisfied, with a certain optimization objective, e.g., minimizing makespan or maximizing throughput. We assume that actor-to-processor mapping is determined at design time, and do not consider any runtime migration and load balancing issues.

There are two types of multiprocessor schedules: for *non-overlapped* schedules, different iterations of the SDF graph execution cannot overlap in time; for *overlapped* or *pipelined* schedules, different iterations of the SDF graph execution can overlap in time. We focus on overlapped schedules, which can achieve higher throughput than non-overlapped schedules by exploiting inter-iteration dependency.

A *Homogeneous Synchronous Dataflow* (HSDF) graph is a special type of SDF graph where all token production and consumption rates are 1. Every SDF graph can be converted to its equivalent HSDF graph [1], but the size of the resulting HSDF graph may be much larger than the original SDF graph. For HSDF graphs, it is well-known that an optimal periodic overlapped schedule with maximum throughput assuming unlimited number of processors can be computed efficiently in polynomial time [2]. Stuijk *et al* [3] presented a technique for finding such an optimal schedule for a general SDF graph without conversion to its equivalent HSDF graph. However, imposing resource constraints (limited number of processors) makes the mapping and scheduling problem NP-complete. When the number of available processors is less than the application's maximum degree of parallelism (the maximum number of actors that are enabled concurrently), we need to use a scheduling strategy to order firings of actors that share the same processor. There are several possible scheduling strategies [4], including static-order, round-robin, Time-Division Multiple Access (TDMA), fixed-priority and dynamic priority scheduling. We refer to [5] for a discussion on the pros and cons of each strategy. In this paper, we adopt *static-order scheduling*, where a cyclic actor firing sequence is defined offline, and the scheduler simply fires each actor in the ordered defined by the sequence. It is a simple strategy that is well-suited for scheduling among actors within the same application (SDF graph), although its lack of flexibility and composability makes it unsuitable for scheduling between multiple applications, for which TDMA scheduling can achieve better performance isolation and composability.

There are two main scheduling policies for SDF graphs: in *fully-static*, or *time-triggered* scheduling, the precise time instant when each actor fires is determined at compile time; in *self-timed* scheduling, each actor fires as soon as its precedence constraints are satisfied and the processor is available. Self-timed scheduling can achieve higher throughput than fully-static scheduling, especially if actor execution times show a large degree of variation. SDF scheduling has the monotonic property [1], i.e., an earlier production of a token cannot result in a later start of an actor during self-timed execution. As a result, there is no scheduling anomaly, i.e., if a self-timed schedule exists that meets the throughput or latency constraints when all actor execution times are equal to their WCET values, then the self-timed schedule with

1052-8725/08 $25.00 © 2008 IEEE
DOI 10.1109/RTSS.2008.49

reduced actor execution times will also meet the throughput or latency constraints. In this paper, we consider all actor execution times to be constant and equal to their WCETs, and do not consider or take advantage of variations in actor execution times. Therefore, self-timed scheduling amounts to fully-static scheduling with the additional property of being work-conserving, i.e., the processor should not be idle when some actor on it is ready to fire.

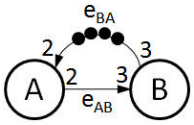
In this paper, we limit our attention to HSDF graphs, and address the following design space exploration and optimization problem:

Given a HSDF graph and a hardware platform consisting of multiple identical processors connected with a communication network with guaranteed latency and constant bandwidth[1]*, find a actor-to-processor mapping and static order schedule on each processor to maximize application throughput.*

This paper is structured as follows: We provide a brief introduction to SDF in Section 2, then explain the key issues involved in actor mapping and static order scheduling in Section 3. We discuss how to allocate buffer space between two adjacent actors mapped to different processors in Section 4. We present the details of the SAT-based techniques in Section 5; performance evaluation results in Section 6; related work in Section 7; conclusions in Section 8.

2 Introduction to SDF

A SDF graph $G = (V, E)$ is a directed graph with a set of nodes $V = \{v_i | 1 \leq i \leq n\}$, representing the actors, and a set of directed edges $E = \{e_{ij} | i, j \in [1, \ldots, n]\}$, representing the communication channels from source actor v_i to sink actor v_j. When an actor v_i fires, the number of tokens it produces (consumes) on each output (input) edge e_{ij} is fixed and known at compile time, denoted as $prod(e_{ij})$ ($cons(e_{ij})$). Each channel e_{ij} has a known capacity in terms of the maximum number of tokens that it can store, denoted as buf_{ij}. Each channel e_{ij} may contain a number of initial tokens, also called *delays*, denoted as d_{ij}. A *simple cycle* is a cycle in the SDF graph where no node appears more than once. Each actor v_i has a known Worst-Case Execution Time (WCET) t_i.



	Init	A	A	B	A	B
e_{BA}	4	2	0	3	1	4
e_{AB}	0	2	4	1	3	0

Figure 1. A SDF graph with schedule $AABAB$.

As an example, Fig. 1 shows a simple SDF graph, and how the number of tokens on the 2 edges changes after each actor firing. Each firing of actor A consumes 2 tokens on edge e_{BA}, and produces 2 tokens on edge e_{AB}; each firing of actor B consumes 3 tokens on edge e_{AB}, and produces 3 tokens on edge e_{BA}. We can view actor A as the producer, and actor B as the consumer, connected by edge e_{AB} with buffer size 4. Number of tokens on edge e_{BA} denotes the number of empty spaces. Or we can view actor B as the producer, and actor B as the consumer, connected by edge e_{BA} with buffer size 4. The two views are symmetric and equivalent from the perspective of SDF theory. A feasible schedule is `AABAB`. Starting from the initial state in Fig 1, if the actors are invoked in the sequence `AABAB`, then the SDF graph goes back to the initial state. Therefore, we can execute this sequence of actor firings repeatedly without any deadlock or buffer overflow conditions. The *repetition count* of actor `A` is 3, and that of `B` is 2. They can be obtained by solving the *balance equation* `repCnt(A)*2=repCnt(B)*3`. The firing of all actors in the SDF graph by their repetition counts is called an *iteration*, and the average execution time of an iteration is called an *iteration period*. A schedule on a multiprocessor system is called *overlapped* or *pipelined* when actor firings from different iterations can overlap with each other in time. We focus on overlapped schedules in this paper, which can achieve higher throughput than non-overlapped schedules. For overlapped schedules, there may be multiple iterations in a single cyclic schedule (a repeating pattern of actor firings), and the iteration period is length of the cyclic schedule divided by the number of SDF graph iterations in the cycle.

A HSDF graph has $prod(e_{ij}) = cons(e_{ij}) = 1$ for all edges e_{ij}. In order to emphasize the difference, we use the term *multirate SDF graphs* to refer to general SDF graphs. The *Cycle Ratio* of a cycle c in a HSDF graph is defined as $CR(c) = \sum_{i \in N(c)} t_i / \sum_{e \in E(c)} d_e$, where $N(c)$ is the set of all nodes traversed by cycle c, and $E(c)$ is the set of all edges traversed by cycle c. The *Maximum Cycle Ratio* (MCR) of a HSDF graph G is defined as:

$MCR(G) = \max_{c \in C(G)} (\sum_{i \in N(c)} t_i / \sum_{e \in E(c)} d_e)$

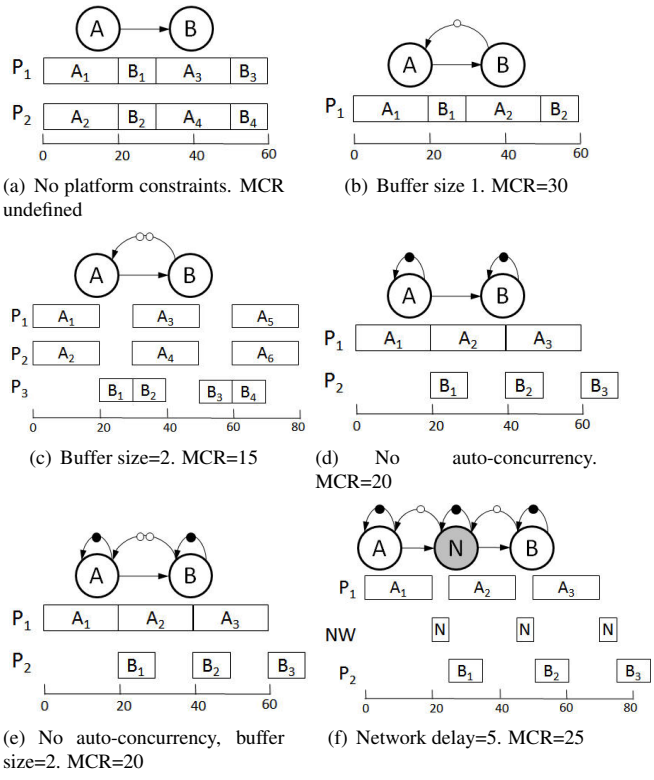
where $C(G)$ is the set of simple cycles in graph G; $N(c)$ is the set of all nodes traversed by cycle c, and $E(c)$ is the set of all edges traversed by cycle c. The cycle with the largest MCR among all simple cycles in graph G is called the *critical cycle*. There may be multiple critical cycles in graph G with the same MCR. It is well-known that the maximum throughput of a HSDF graph without resource constraints (with unlimited number of processors available) is the inverse of its MCR, i.e., the minimum iteration period is equal to its MCR, hence maximizing throughput of a HSDF graph is equivalent to minimizing its MCR. However, the minimum iteration period of a HSDF graph may be larger than its MCR if there are resource constraints, i.e., the number of processors is smaller than the degree of parallelism of the HSDF graph, so that some actor firings must be serialized on a single processor. If a HSDF graph does not contain a cycle, then its MCR is undefined, and its throughput can be increased infinitely by adding more processors, e.g., the example in Fig. 2(a). There are efficient polynomial-time algorithms for computing the MCR of a HSDF graph [6]. A *deadlock cycle* is a simple cycle with no initial tokens.

3 Issues in Actor Mapping and Scheduling

3.1 Adding Platform Constraints to HSDF Graph

We adopt the approach proposed in [4] to model platform constraints, including buffer size constraints, network delay and pro-

[1] this can be achieved with a hard real-time Network-on-Chip, for example.



(a) No platform constraints. MCR undefined

(b) Buffer size 1. MCR=30

(c) Buffer size=2. MCR=15

(d) No auto-concurrency. MCR=20

(e) No auto-concurrency, buffer size=2. MCR=20

(f) Network delay=5. MCR=25

Figure 2. Effect of platform constraints on application throughput. WCET(A)=20; WCET(B)=10; WCET(N)=5. White tokens denote empty space. Subscripts denote the iteration number if each actor firing.

cessor sharing, by adding additional actors and/or edges to the SDF graph. This bears some similarity to Model-Driven Architecture (MDA) [7] for enterprise application development, where a *Platform-Independent Model (PIM)* that contains only business logic is automatically transformed into a *Platform-Specific Model (PSM)* by adding modeling elements specific to a given middleware platform, so that developers only need to focus on the PIM, and rely on automated tools to generate a different PSM for a new middleware platform. However, our focus is on design space exploration and optimization while mapping an application to platform, which is very different from MDA's goal of improving application portability and maintainability in the face of multiple and changing middleware standards. Nonetheless, we borrow from the MDA terminology, and refer to the application SDF as the PIM, and the SDF enhanced with platform constraints as the PSM, which may contain additional actors and edges to model platform elements. We distinguish between *application actors* that are common to the PIM and PSM, and *network actors* modeling network delays that are in the PSM only. We draw network actors as shaded circles to distinguish them from application actors. There may be multiple PSMs for the same PIM with different levels of detail.

Stuijk *et al* [4] used the terms *Application SDF Graph*, which corresponds to the PIM; *Memory-Aware SDF Graph*, which corresponds to the PSM with buffer size constraints; and *Binding-Aware SDF Graph*, which corresponds to the PSM with buffer size constraints and network delays. In this paper, we assume that the buffer sizes on all edges have been determined before mapping and scheduling, possibly using the algorithm in [3] or [8] for determining a minimum buffer size distribution to guarantee a given throughput constraint. We start from a PSM (Memory-Aware SDF Graph), and perform mapping and scheduling to obtain another PSM (Binding-Aware SDF graph) with the maximum possible throughput, which is almost always lower than the PSM before mapping and scheduling.

We illustrate this PIM to PSM transformation procedure with examples. Even though we focus on HSDF graphs, the same procedure are also applicable to multirate SDF graphs *except the static order schedule constraints*, which cannot be represented by adding additional edges in the multirate SDF graph, but can only be enforced by constraints on SDF firing rules [3]. Fig. 2 illustrates the effect of platform constraints on application throughput (inverse of MCR). Note that the schedules in the figure start at time 0, while throughput should be calculated based on the steady-state execution, ignoring the initialization stage. For clarity, we use black tokens to denote the application data tokens, and white tokens to denote empty space, although this makes no difference from the perspective of SDF theory. Fig. 2(a) shows an application HSDF graph without any platform constraints, and one possible schedule with 2 processors. Since there are no cycles in the graph, MCR is undefined, hence the achievable throughput is unlimited, and increases with additional processors. This is a so-called "embarrassingly parallel" application in the parallel computing literature, where adding more processors always results in higher throughput. The HSDF graph in Fig. 2(b) has buffer size constraint of 1 between actors A and B, resulting in the critical cycle AB with MCR=30. This constraint serializes execution of actors A and B, so that adding more processors does not help increase throughput. The HSDF graph in Fig. 2(c) has buffer size constraint of 2 between actors A and B, so the critical cycle AB has MCR=15. Due to increased buffer size, execution of actors A and B can overlap. The schedule in Fig. 2(c) utilizes 3 processors to finish 2 iterations of application execution every 30 time units, with a throughput of 1/15. The HSDF graph in Fig. 2(d) has no buffer size constraint, but does not allow auto-concurrency for actors A and B, i.e., one firing of an actor must finish before its the next firing can begin. This constraint specifies that each actor can only be assigned to at most 1 processor, and cannot be replicated on multiple processors to increase throughput, as in Figs. 2(a) and 2(c). The critical cycle is the self-cycle of actor A with MCR=20. Fig. 2(e) shows that adding a buffer size constraint of 2 does not affect throughput, since the critical cycle is still the self-cycle of actor A. (Note that adding a buffer size constraint of 1 will change the critical cycle to AB with MCR=30, same as Fig. 2(b)). The HSDF graph in Fig. 2(f) adds network actor N with WCET 5 to model network delay. We add such a network actor between any two actors connected by an edge in the application HSDF graph and mapped onto different processors, so there is no network contention between different communicating actor pairs. There is a self-edge on actor N to prevent auto-concurrency, which means transmitting 2 tokens

takes twice as long as transmitting 1 token. The critical cycle is AN with MCR=25. If auto-concurrency is allowed, then an actor can be duplicated on multiple processors for data parallel execution, and *unfolding* [2] can be exploited to increase application throughput, which helps to increase the degree of concurrency with the cost of increased code size. Whether auto-concurrency is feasible for a given actor is determined by its internal algorithmic implementation: it is feasible for *stateless* actors, but not feasible for *stateful* actors with inter-iteration dependency caused by maintaining and updating internal state information at each iteration. In this paper, we make the simplifying assumption that there is no auto-concurrency, so each actor in the SDF graph can be mapped to a single processor only. Therefore, we assume there is always a self-edge with 1 token delay on each actor, even when it is omitted from the figures. As a result, we do not need to consider any unfolding operations, since they do not help to increase application throughput. (The same assumption is made in [4].)

3.2 Static Order Scheduling

For a HSDF graph, each actor fires once at one iteration, so imposing a static order schedule on the same processor is equivalent to imposing a total actor ordering, and can be expressed by adding additional precedence edges in the HSDF graph [9]. (This is not possible in general for multirate SDF graphs, as discussed in [5]). Assuming the static order schedule is $A_1A_2 \dots A_n$, then we add a cycle of directed edges $((A_1, A_2), (A_2, A_3), \dots, (A_{n-1}, A_n), (A_n, A_1))$ with an initial token on the edge (A_n, A_1) to ensure that actor A_1 is the first actor that can fire.

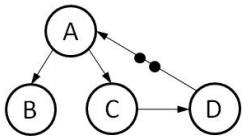
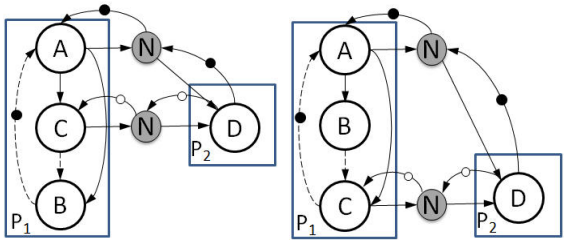


Figure 3. Platform-independent HSDF graph. WCET(A)=WCET(B)=WCET(C)=5.



(a) HSDF graph with static order schedule ACB on P_1, with MCR=17.5 (cycle ACD)

(b) HSDF graph with static order schedule ABC on P_1, with MCR=20 (cycle $ABCD$)

Figure 4. Two different static order schedules on P_1. WCET(A)=WCET(B)=WCET(C)=5, WCET(N)=10.

We use an example adapted from [10] to demonstrate the effect of static order scheduling on the MCR. Fig. 3 shows a PIM HSDF graph without any buffer size or processor mapping constraints, with MCR=7.5 (cycle ACD). Fig. 4 shows two different static order schedules on P_1 with different resulting MCR values. For clarity and to avoid clutter, we assume that edges internal to P_1 have infinite buffer size, and omit the back edges. Edges connecting to the network actors have buffer size 1. We also omit the implicit self-edge with a single delay on each actor to prevent auto-concurrency. We use dotted edges to encode the static order schedule based on the algorithm in [9]. The platform independent HSDF graph in Fig. 3 does not capture the mutual exclusion between firings of actors B and C due to processor sharing, so it allows concurrent firing of actors B and C. At runtime, there may be multiple possible actor firing orders on P_1. Adopting the schedule ACB on P_1 results in a HSDF graph with MCR=17.5 (Fig. 4(a)), while adopting the schedule ABC on P_1 results in a HSDF graph with MCR=20 (Fig. 4(b)). If we do not impose a static order schedule on P_1, then we have no control over the runtime order between actors B and C, which may be different at different iterations of the HSDF's execution depending on internal implementation of the actors (which output message is sent out first) and the OS scheduling policy. Therefore, we must make the worst-case assumption and calculate the MCR for a given actor-to-processor mapping as the largest MCR value among all possible static order schedules on each processor. Therefore, we should impose the static order schedule that results in the minimum MCR on each processor.

4 Optimal Buffer Allocation

In this section, we present two theorems for optimizing allocation of the total available buffer space between two adjacent actors in the application HSDF graph if they are mapped to different processors to maximize throughput of the resulting PSM. Without these theorems, we would have to search all possible buffer allocations, which creates another dimension in the design space.

Theorem 1. *Consider a HSDF graph where there is a directed edge e_{ij} with tk_{ij} number of data tokens from actor v_i to actor v_j, and another back edge e_{ji} with sp_{ji} number of tokens from actor v_j to actor v_i denoting the empty space. Then the total buffer space of the communication channel from v_i to v_j is $B_{ij} = tk_{ij} + sp_{ji}$. We assume $B_{ij} \geq 2$. Actor v_i has WCET t_i, actor v_j has WCET t_j. If v_i and v_j are mapped to two different processors, modeled by the HSDF graph with an additional actor N with WCET t_n. Using shorthand notations $t_{in} = t_i + t_n$, $t_{jn} = t_j + t_n$, and $t_{ijnn} = t_i + t_j + t_n + t_n$, the optimal buffer allocation (B_{in} between v_i and N, B_{jn} between v_j and N) is the solution to the following integer optimization problem:*

$$\begin{aligned} &\textit{Minimize: } max(t_{in}/B_{in}, t_{jn}/B_{jn}) \\ &\textit{subject to: } B_{in} + B_{jn} = B_{ij} \end{aligned} \tag{1}$$

where B_{in} and B_{jn} are integer variables.

The optimization problem has a closed form solution described

in Equations (2) to (5).

$$B_{in} = \lceil B_{ij} * t_{in}/t_{ijnn} \rceil$$
$$B_{jn} = \lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor \quad (2)$$

if

$$\frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor} \leq \frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor} \quad (3)$$

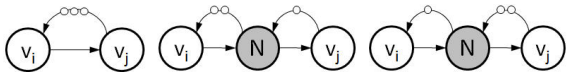
Or:

$$B_{in} = \lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor$$
$$B_{jn} = \lceil B_{ij} * t_{jn}/t_{ijnn} \rceil \quad (4)$$

if

$$\frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor} > \frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor} \quad (5)$$

The proof is shown in Appendix I.



(a) MCR=10 (b) MCR=15 (c) MCR=25

Figure 5. HSDF buffer allocation example. $t_i = 20, t_j = 10, t_n = 5$**.**

Intuitively, Theorem 1 states that buffer space allocation should be biased towards the side of the actor with larger execution time, and the ratio of B_{in} to B_{jn} should be roughly equal to t_{in}/t_{jn}. If the network delay is negligible ($t_n = 0$), then the ratio is roughly equal to t_i/t_j; if the network delay is very large ($t_n = \infty$), then the ratio is roughly equal to 1, i.e., equal distribution.

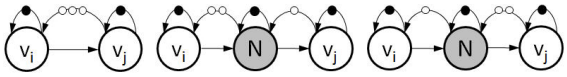
Note that Theorem 1 requires the edge buffer size to be not smaller than 2, since it is not permissible to allocate 0 buffer size to either side of the network actor, which results in a deadlock cycle. As a corollary, if there is only a single buffer space between two adjacent actors A and B, then they must be mapped to the same processor.

As an example, consider the HSDF graph in Fig. 5(a), where $t_i = 20$, $t_j = 10$, $t_N = 5$, B_{ij}=3, and MCR=10. If actors t_i and t_j are mapped to two different processors, then we need to decide how to allocate the total buffer size of 3 to the two sides of sides of the network actor. Fig. 5(b) shows the result of allocating $B_{in} = 2, B_{jn} = 1$, with MCR=max(25/2, 15/1)=15; Fig. 5(c) shows the result of allocating $B_{in} = 1, B_{jn} = 2$, with MCR=max(25/1, 15/2)=25. Therefore, the buffer allocation in Fig. 5(b) is better than that in Fig. 5(c), and can be derived from Theorem 1. First, we know that $t_{in} = t_i + t_N = 25$, $t_{jn} = t_j + t_N = 15$, $t_{ijnn} = t_i + t_j + 2 * t_N = 40$. Since $\frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor} = \frac{15}{\lfloor 3*15/40 \rfloor} = 15 \leq \frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor} = \frac{25}{\lfloor 3*25/40) \rfloor} = 25$, we apply Equation (2) to get:

$B_{in} = \lceil B_{ij} * t_{in}/t_{ijN} \rceil = \lceil 3 * 25/40 \rceil = 2$

$B_{jn} = \lfloor B_{ij} * t_{jn}/t_{ijN} \rfloor = \lfloor 3 * 15/40 \rfloor = 1$

Theorem 1 is concerned with a local optimization problem of deriving an optimal buffer distribution over a single edge, and its correctness proof relies solely on the local properties of actors v_i, v_j and N. Therefore, it is still valid in the face of other possible edges that may involve actors v_i, v_j and N. For example, Fig. 6 shows that adding self-edges to the three actors changes the MCR values, but the buffer allocation of $B_{in} = 2, B_{jn} = 1$ is still the optimal one.



(a) MCR=20 (b) MCR=20 (c) MCR=25

Figure 6. HSDF buffer allocation example with self-edges. $t_i = 20, t_j = 10, t_n = 5$**.**

5 SAT for HSDF Mapping and Scheduling

5.1 Introduction to the SAT DPLL Algorithm

Satisfiability (SAT) solving is a well-known NP-complete problem of assigning values to a set of boolean variables to make a propositional logic formula true. SAT has found wide applicability in Electronic Design Automation (EDA) for solving various constraint satisfaction and formal verification problems. There are many algorithms used for SAT solving, but most algorithms are variants or enhancements of the Davis, Putnam, Logemann and Loveland (DPLL) algorithm. We provide a brief introduction to the basic DPLL algorithm in this section. First, we introduce some basic terminology. The SAT formula is typically written in Conjunctive Normal Form (CNF) consisting of a conjunction of multiple disjunctions. $X = x_1, \ldots, x_n$ is a set of boolean variables. A *literal* is either a variable x_i or its negation $\overline{x_i}$. A *clause* is a disjunction of literals. An *assignment* is a set of literals that does not contain any variables. An assignment of size n is called a *complete assignment*, otherwise it is called a *partial assignment*. An assignment *satisfies* a clause if the clause is true under the assignment, otherwise it *falsifies* the clause. A boolean formula $\mathcal{F}$ in *Conjunctive Normal Form* (CNF) is conjunction of a set of clauses. A clause of size 1 is called a *unit clause*. When a formula contains a unit clause with a single literal l, it can be simplified by removing all clauses that contain the literal l, and removing $\bar{l}$ from all the clauses that contain it. This is called *Unit Propagation*. As an example, consider the CNF formula $(\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_4} \vee \overline{x_5})$. The complete assignment $x_1, x_2, x_3, \overline{x_4}, x_5$ is a satisfying assignment, since it makes all three conjunctive clauses true.

Algorithm 1, excerpted from [11], illustrates the basic DPLL algorithm as implemented in MiniSat [12]. (Note that this is the *iterative* version of the DPLL algorithm, which may look different from the *recursive* version in other literature.) It performs a systematic depth-first search on all possible boolean variable assignments. At the tree root, no variables are assigned. The propagation queue Q contains all literals pending propagation, initialized to the empty queue at Line 1. At each step, a variable is chosen and assigned to be true or false, to explore two branches of the search tree (Line 13). Values of some other variables may

Algorithm 1: Basic DPLL algorithm with conflict driven learning and non-chronological backjumping.

```
   Output: result: boolean
1  Q=φ;
2  while true do
3      UnitPropagation();
4      if Conflict then
5          AnalyzeConflict();
6          if TopLevelConflict then
7              return false;
8          else
9              LearnClause(), Backjump();
10     else if All variables assigned then
11         return true;
12     else
13         l := SelectLiteral(), Enqueue(Q, l);
```

be derived based using Unit Propagation (Line 3). If that leads to a conflicting clause, we analyze the conflict, and if it is at the top level of the search tree (Line 6), the formula is unsatisfiable, and the algorithm ends. Otherwise, the procedure backtracks using non-chronological backjumping and clause learning (Line 8). (*Chronological backtracking* refers to going back one level up the search tree, while *non-chronological backjumping* refers to jumping up multiple levels of the search tree to the variable assignments that caused the conflict.)

	Q	Propagated.Lit.	Remaining Clauses
1	x_1	ϕ	$(\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_4} \vee \overline{x_5})$
2	x_2, x_3	x_1	$(\overline{x_4} \vee \overline{x_5})$
3	ϕ	x_1, x_2, x_3	$(\overline{x_4} \vee \overline{x_5})$
4	$\overline{x_4}$	x_1, x_2, x_3	$(\overline{x_4} \vee \overline{x_5})$
5	ϕ	$x_1, x_2, x_3, \overline{x_4}, x_5$	1

Table 1. Example to illustrate the basic DPLL algorithm.

We use an example to illustrate the basic DPLL algorithm without conflict driven learning and non-chronological backtracking, and refer the interested reader to [11] for more details. Consider the boolean formula $(\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_4} \vee \overline{x_5})$. Table 1 shows the running process of the basic DPLL algorithm. The leftmost column denotes the iteration step number; Q denotes the propagation queue containing the pending literals to be propagated; *Propagated.Lit.* denotes the literals that have already been propagated. Suppose we initially select x_1 as the literal to be propagated and put it in the propagation queue Q, i.e., set $x_1 = 1$. After propagation, we can infer that $x_2 = 1, x_3 = 1$ from the clauses $(\overline{x_1} \vee x_2)$ and $(\overline{x_1} \vee x_3)$, so we put the literals x_2, x_3 into Q (Step 2). After propagation, we cannot infer any new facts, so no new literals are added to Q (Step 3). Next, we select $\overline{x_4}$ as the literal to be propagated and put it in Q, i.e., set $x_4 = 0$. After propagation, we can infer that $x_5 = 1$ from the clause $(\overline{x_4} \vee \overline{x_5})$ (Step 5). At this point, we have found a satisfying assignment of $\{x_1, x_2, x_3, \overline{x_4}, x_5\}$. Note that this is not the only possible solution, e.g., $\{x_1, x_2, x_3, x_4, \overline{x_5}\}$ is another solution if we had selected x_4 instead of $\overline{x_4}$ as the literal to be propagated, i.e., set $x_4 = 1$ instead of $x_4 = 0$. This example does not have any backtracking, since no conflicts are encountered during propagation.

5.2 High-Level Optimization Approaches

Even though SAT is traditionally used as a decision procedure that returns a yes/no answer of satisfiability and a satisfying assignment if yes, it can also be used for optimization problems to find a satisfying assignment that optimizes a given objective function. One example is the Maximum Satisfiability problem (MAX-SAT) that asks for the maximum number of clauses which can be satisfied by any assignment, which can be achieved by integrating the *Branch-and-Bound* algorithm into the SAT solver [11]. Branch-and-Bound is a well-known optimization technique that consists of two parts: the first part is a *branching* procedure that constructs a search tree in a depth-first manner, and the second part is a *bounding* procedure that prunes a tree branch at a tree node n representing a partial solution, if it can be established that value of the objective function to be minimized at node n has a lower bound that is larger than the best objective function value seen so far. To integrate Branch-and-Bound into the SAT solver [11], instead of exiting and returning a satisfying assignment when one is found, the algorithm records the value of the objective function of all satisfying assignments seen so far, and continue exploring the search tree.

Instead of modifying the SAT solver, we can use *Benders Decomposition*. *Classic Benders Decomposition* (CBD) divides the optimization problem into a *master problem* with a set of primary variables X, and a sub-problem with a set of secondary variables Y while holding the primary variables at fixed values $\overline{X}$. Solution of the sub-problem generates the *Benders Cut*, additional constraints that the primary variables X must satisfy, in the form of a linear inequality based on Lagrangian multipliers obtained from a solution of the sub-problem dual. The process iterates until the master problem and the sub-problem converge in their assignments to the variables in X and Y. CBD requires that the sub-problem to be a continuous linear programming problem, which is not true for scheduling problems. Hooker *et al* [13] introduced *Logic-Based Benders Decomposition* (LBBD) in the context of logic circuit verification, which shares the same general framework as CBD, except that the sub-problem is not constrained to be a linear programming problem, but can be any combinatorial optimization problem, and generation of the *no-good* solution as the Benders cut is application-specific, instead of always the solution of the sub-problem dual as in CBD.

Fig. 7 shows two optimization approaches. In the approach labeled `LBBD` in Fig. 7(a), we decompose the problem into a master problem (actor-to-processor mapping) and a sub-problem (deadlock detection + MCR computation). At each iteration, the master problem SAT solver provides a candidate solution of actor-to-processor mapping and ordering on each processor to the sub-problem solver, which then performs deadlock detection and calculates the MCR, and feeds back the Benders cut, in this case, *blocking clauses* learned from the deadlock or critical cycle, to the master problem SAT solver to prevent generation of any actor mapping and ordering that contain the same deadlock/critical

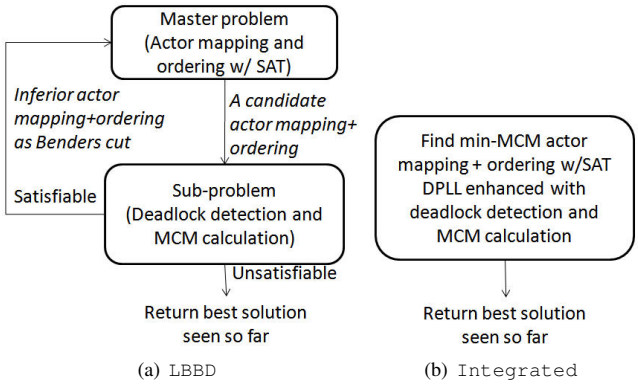


Figure 7. Two alternative optimization frameworks.

cycles. The procedure iterates until no feasible solutions can be found, and the best solution seen so far with the minimum MCR is returned as the optimal solution. In the approach labeled `Integrated` in Fig. 7(b), we enhance the SAT DPLL procedure with deadlock detection and MCR computation algorithms to perform Branch-and-Bound optimization within the SAT solver. In the LBBD approach, a complete solution of actor mapping and ordering is generated before invoking the sub-problem solver, while in the `Integrated` approach, the search tree can be pruned early using Branch-and-Bound when a partial solution is generated.

5.3 The `LBBD` Approach

The boolean encoding is similar to [14], which addresses the problem of mapping a task graph to a multiprocessor platform to minimize makespan. We define the following boolean variables:

$$\forall v \in V, \forall p \in P, x_a(v,p) = \begin{cases} 1 & \text{if actor } v \text{ is mapped to CPU } p \\ 0 & \text{otherwise} \end{cases}$$

$$\forall (v_i, v_j) \in E, x_c(v_i, v_j) = \begin{cases} 1 & \text{if actors } v_i \text{ and } v_j \text{ are} \\ & \text{mapped to different PEs} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall v_i, v_j \in V, (v_i, v_j) \notin E^T, (v_j, v_i) \notin E^T, x_c(v_i, v_j) = 0,$$

$$x_d(v_i, v_j) = \begin{cases} 1 & \text{if actor } v_i \text{ precedes actor } v_j \\ & \text{on the same processor} \\ 0 & \text{otherwise} \end{cases}$$

Variables x_a denote mapping of actors to processors; variables x_c denote inter-processor communication between two actors mapped to different processors; variables x_d denote ordering relationships between pairs of actors mapped onto the same processor that are not precedence-constrained in the application graph. E^T is the transitive closure of the original edges E in the application graph, so two actors are not precedence-constrained when $(v_i, v_j) \notin E^T, (v_j, v_i) \notin E^T$. The x_c variables can be inferred from the x_a variables, but they are added to allow a more compact constraint encoding.

$$\forall v \in V, \vee_{p \in P} x_a(v,p) \Rightarrow 1 \tag{6}$$

$$\begin{aligned} &\forall v \in V, \forall p_1, p_2 \in P, p_1 \neq p_2, \\ &x_a(v,p_1) \wedge x_a(v,p_2) \Rightarrow 0 \end{aligned} \tag{7}$$

$$\begin{aligned} &\forall (v_1, v_2) \in E, \forall p_1, p_2 \in P, p_1 \neq p_2, \\ &x_a(v_1,p_1) \wedge x_a(v_2,p_2) \Rightarrow x_c(v_1,v_2) \wedge \neg x_d(v_1,v_2) \wedge \neg x_d(v_2,v_1) \end{aligned} \tag{8}$$

$$\begin{aligned} &\forall (v_1, v_2) \in E, \forall p \in P, \\ &x_a(v_1,p) \wedge x_a(v_2,p) \Rightarrow \neg x_c(v_1,v_2) \end{aligned} \tag{9}$$

$$\begin{aligned} &\forall v_1, v_2 \in V, (v_1,v_2) \notin E_0^T, (v_2,v_1) \notin E_0^T, \forall p \in P, \\ &x_a(v_1,p) \wedge x_a(v_2,p) \Rightarrow x_d(v_1,v_2) \vee x_d(v_2,v_1), \\ &x_d(v_1,v_2) \wedge x_d(v_2,v_1) \Rightarrow 0 \end{aligned} \tag{10}$$

Equations (6) to (10) encode the constraints among the boolean variables x_a, x_c and x_d. $\Rightarrow$ is the logical implication operator. Equations (6) and (7) specify that an actor is mapped to 1 and only 1 processor; Equation (8) specifies that if two actors v_1 and v_2 connected by an edge are mapped to two different processors, then a communication cost is incurred ($x_c(v_1,v_2) = 1$), and there is no precedence constraint between them ($\neg x_d(v_1,v_2) \wedge \neg x_d(v_2,v_1)$); Equation (9) specifies that if two actors v_1 and v_2 connected by an edge are mapped to the same processor, then there is no communication cost ($x_c(v_1,v_2) = 0$); Equation (10) specifies that if two actors v_1 and v_2 mapped to the same processor, and there is no edge between them in the transitive closure of the application graph, then an ordering edge should be inserted between them.

The sub-problem solver is the deadlock cycle detection and MCR computation algorithm. Each time a deadlock cycle or a critical cycle is detected, it is returned to the master problem solver as a *Benders Cut* to prevent generation of the same cycles in the future. Let V_c and E_c denote the set of actors and edges in a deadlock cycle or critical cycle c, respectively. The following blocking clause is added to the SAT solver:

$$\left(\left(\bigwedge_{\substack{v \in V_c - V, \\ (v_i,v),(v,v_j) \in E_c}} x_c(v_i,v_j)\right) \wedge \left(\bigwedge_{\substack{v_k,v_h \in V, \\ (v_k,v_h) \in E_c - E}} x_d(v_k,v_h)\right) \Rightarrow 0\right. \tag{11}$$

In the first conjunctive term of Equation (11), actor $v \in V_c - V$ is an actor in the cycle that models network delay in the HSDF graph G' after mapping/scheduling, since it is not in the collection of nodes V in the HSDF graph G before mapping/scheduling. If there are two edges (v_i, v) and (v, v_j) in the cycle in G', then actors v_i and v_j are mapped to different processors, hence $x_c(v_i,v_j) = 1$. In the second term, v_k and v_h are actors in G; (v_k, v_h) is an edge in the cycle in G', but it is not part of G. This means that the edge (v_k, v_h) models an added precedence relationship from v_k to v_h as part of the static order schedule on the same processor. Equation (11) ensures that at least one of the boolean variables in the clause is false, thus preventing the cycle c from being generated again in the future during search process.

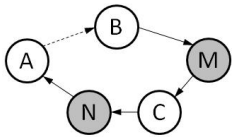


Figure 8. Example to illustrate blocking clauses added due to a deadlock or critical cycle.

For example, if a deadlock or critical cycle c is detected in the PSM consisting of 5 actors $ABMDN$, as show in Fig. 8, where we have omitted any tokens on the edges in case of a critical cycle. The edge AB is added to the PSM due to actor ordering constraints. To prevent the same cycle in Fig. 8 from being generated again in the future, the following clause is added to the SAT solver:

$$(x_c(B,C) \wedge x_c(C,A) \wedge x_d(A,B)) \Rightarrow 0$$

which means that at least one of the boolean variables among $x_c(B,C)$, $x_c(C,A)$ and $x_d(A,B)$ must be 0.

5.4 The `Integrated` Approach

We first present Theorem 2, which forms the basis for the branch-and-bound procedure for early search tree pruning at non-leaf nodes.

Theorem 2. *MCR of a PSM $G_{partial}$ with a partial solution of actor mapping and ordering where some actors are not yet mapped to processors, or are mapped but not yet ordered, forms a lower bound on the MCR of any final PSM G_{full} obtained from extending $G_{partial}$ to a full solution of actor mapping and ordering.*

Proof. Consider any fully mapped and scheduled HSDF graph G_{full} obtained by mapping and/or scheduling additional actors on the basis of a partially mapped and scheduled HSDF graph $G_{partial}$. The set of actors V in $G_{partial}$ is a subset of the set of actors V' in G_{full}, which can contain additional actors representing network delay. Consider any cycle c in $G_{partial}$. If no new network actors are introduced into cycle c during any of the mapping steps leading to G_{full}, then G_{full} also contains the cycle c with the same actors and edges as cycle c in $G_{partial}$, hence $CR(c') = CR(c)$. If one or more new network actors are introduced into cycle c during one of the mapping steps leading to G_{full}, then G_{full} contains a new cycle c' with more actors and edges than cycle c in $G_{partial}$, but not more tokens, since we never add any new tokens to the HSDF graph at any of the mapping and scheduling steps. Hence $CR(c') \geq CR(c)$. In addition, G_{full} may contain many new cycles that are not present in $G_{partial}$ due to the newly added actors and edges encoding the mapping and ordering constraints. Therefore, $MCR(G_{full}) \geq MCR(G_{partial})$ based on the definition of MCR. □

Algorithm 2 shows the enhanced DPLL algorithm for finding the optimal actor mapping and ordering with minimum MCR. Lines 4 to 10 are essentially the same as the basic DPLL Algorithm 1, except that if a top-level conflict is detected, we have found the optimal solution, and return the optimal MCR value

Algorithm 2: DPLL algorithm with Branch-and-Bound search to minimize the MCR.

```
   Output: MCRBest : float
   Output: actor mapping/ordering
1  InitializeQueue();
2  float MCRBest=0;
3  while true do
4      UnitPropogation();
5      if Conflict then
6          AnalyzeConflict();
7          if TopLevelConflict then
8              return MCRBest + actor mapping/ordering;
9          else
10             LearnClauseDueToConflict(), Backjump(), goto 3;
11     thePSM=ConstructPSM();
12     boolean Deadlocked=CheckDeadlock(thePSM);
13     if Deadlocked then
14         LearnClauseDueToDeadlock(), ChronologicalBacktrack(),
           goto 3;
15     float MCRLB=CalculateMCR(thePSM);
16     if MCRLB ≥ MCRBest then
17         LearnClauseDueToCriticalCycle(),
           ChronologicalBacktrack(), goto 3;
18     if All variables assigned then
19         MCRBest=min(MCRBest, MCRLB);
20         LearnClauseDueToCriticalCycle(), RestartSearch();
21     else
22         l := SelectLiteral(), Enqueue(Q, l);
```

plus the corresponding actor mapping and ordering. As the algorithm runs, more and more blocking clauses are added as constraints so that Line 8 is the only legal point of return. At Line 15, the function ConstructPSM() constructs a *thePSM* where some(or all) of the actors have been mapped to processors, and some (or all) of the actor orderings on each processor haven been established. The function CheckDeadlock() checks to see if there exists any deadlock cycles in *thePSM*. If yes, then we learn some additional clauses from the deadlock cycle with Equation 11, and backtrack to the previous search tree level by removing the mapping of the most recent actor that was mapped (Line 14). If there is no deadlock, then function CalculateMCRLB() is used to calculates MCR of *thePSM* as the lower-bound of all possible HSDFs where all actors have been mapped to processors by extending the current partial (or full) actor-to-processor mapping modeled by *thePSM*. If the current MCR lower-bound is not smaller than the best MCR of previously-seen full actor-to-processor mapping so far, then we learn some additional clauses from the Critical Cycle with Equation 11, and backtrack to the previous search tree level by removing the mapping of the most recent actor that was mapped (Line 17). At Lines 18 and 20, if all boolean variables have been assigned, we have reached the leaf level of the search tree, and found a satisfying assignment for the SAT problem. In contrast to the basic DPLL algorithm, we do not stop here, since our goal is to find the satisfying assignment with the minimum MCR. Instead, we remember the best MCR seen so far, and restart the search from the top level of the search tree. As the algorithm

proceeds, more and more blocking clauses are added to prevent generation of the deadlock and critical cycles seen so far. Eventually the boolean formula will become infeasible, and the procedure finishes at Line 8.

Satisfiability Modulo Theories (SMT) [15] extends the SAT DPLL procedure to DPLL(T) (T stands for theory) by adding the ability to handle some decidable theories, e.g., equality with uninterpreted function symbols, linear arithmetic over integers and rationals. There are several integration strategies between the SAT solver and the theory solver [15]: for *very lazy* integration, the theory solver does not feedback any information to the SAT solver; for *lazy integration*, the theory solver returns a blocking clause (an explanation) to the SAT solver whenever the theory solver detects that the problem is infeasible; for *eager integration*, the theory solver actively participates in value propagation and conflict analysis. The `LBBD` approach can be viewed as "Satisfiability Modulo SDF Theory" (SMS) with the lazy integration strategy, while the `Integrated` approach can be viewed as SMS with the eager integration strategy, where the theory solver can be invoked on a partial boolean variable assignment to generate blocking clauses. One key difference from SMT is that we use Branch-and-Bound to optimize an objective function (minimum MCR), while SMT is for satisfiability checking only instead of optimization (although there has been recent work on SMT solvers with optimization capability using binary search [15]).

Next, we discuss a number of tuning knobs for improving performance.

Invocation Frequency of Theory Solver We use the term *theory solver* to refer to the deadlock cycle detection and MCR computation algorithm, using the terminology from SMT. If the theory solver is invoked only at the leaf-level of the search tree, then the `Integrated` approach degenerates to the `LBBD` approach. If the theory solver is invoked after every single boolean variable assignment, as shown in Algorithm 2, then many invocations may be useless and do not learn any new clauses, or learn very few of them. We adopt the approach of invoking the theory solver once after N boolean variables are assigned, where N is a configurable variable chosen with trial-and-error. We set N to be the number of actors as a simple heuristic in the performance experiments.

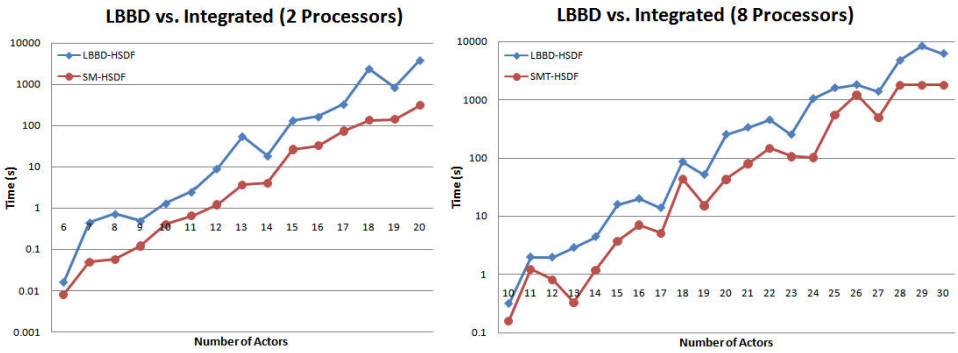
Branching Heuristic An important design choice of the DPLL algorithm is the branching heuristic in the function SelectLiteral(), which selects a boolean variable and assigns it a true or false value, then performs propagation to infer the values of other variables. The order of assignment of the boolean variables in the SAT solver has a large impact on search tree size and search efficiency. MiniSat has its built-in dynamic branching heuristic that is quite effective in solving general SAT problems. In our case, we need to decide which actor to be mapped to a processor by assigning variables x_a and x_c, and which precedence edge should be added between two actors without precedence constraints on the same processor by assigning variables x_d. Instead of using dynamic branching heuristic built into MiniSat, we exploit application domain knowledge to design a static branching heuristic: whenever two actors (v_1 and v_2) are mapped to the same processor and there is no dependency relationship between them (the precondition in Equation (10) is satisfied), then immediately impose a static order by adding a precedence edge between them, i.e.,assign the boolean variables $x_d(x_1, x_2)$ and $x_d(x_2, x_1)$. This amounts to interleaved mapping and scheduling, instead of performing all mapping before scheduling.

Number of Clauses Learned Each invocation of the theory solver may generate one to many blocking clauses. We must balance the cost of generating new clauses against their utility in pruning the search space. We can choose to return all deadlock cycles and critical cycles with cycle ratio greater than or equal to the MCR of a known solution, but it is very computationally expensive or infeasible to traverse all cycles in the graph, since the number of cycles grows exponentially with the graph size. We choose to return the first deadlock cycle detected, or all critical cycles whose cycle ratio is equal to the MCR of the current partial solution, which can be efficiently computed with known algorithms [6]. (This issue is also present in the LBBD approach, where the same rules are applied.)

6 Performance Evaluation

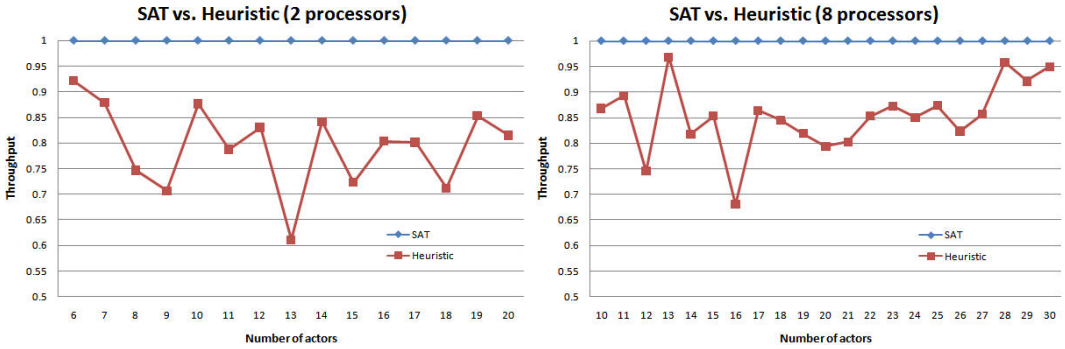
We use the software tool SDF^3 [16] to generate random HSDF graphs with the following input parameters: WCET of each application actor ranges from 10 to 50; WCET of each network actor is 5, i.e., inter-processor communication time is 5 time units per token; buffer size on each edge ranges from 1 to 5 tokens; the probability of an edge to have initial tokens is 0.2. After obtaining a HSDF graph from SDF^3, we insert a back edge for every forward edge in the HSDF graph to model edge buffer size constraints. The experiments are run on a Linux workstation with an Intel dual-core 2.4GHz 64-bit processor and 4GB of main memory. We use a utility program *Memtime* to measure the running time of the overall optimization algorithm.

We first compare the relative performance of `LBBD` and `Integrated` in terms of algorithm running time. Algorithm running time depends on the search space size, which is in turn dependent on many problem-specific factors. In addition to the number of actors and processors, the HSDF graph shape, size (number of edges) and token distribution can also have a large impact. Therefore, it is not entirely fair to use the number of actors as the sole measure of scalability. Nonetheless, the number of actors serves as a rough metric to gauge performance of the optimization algorithms, since the number of possible actor mapping/orderings grows exponentially with the number of actors. Fig. 9 shows algorithm running times for `LBBD` and `Integrated` for different number of actors. Since the difference in algorithm running time can be quite large, the y-axis is drawn with a log scale. Having more processors helps reduce the running time, since the main performance bottleneck turns out to be static order scheduling, and having more processors reduces the total number of possible static order schedules to be explored. Results show that the `Integrated` approach consistently outperforms the `LBBD` approach significantly. This is not surprising since the `Integrated` approach invokes the theory solver on partial solutions to return blocking clauses and achieve more effective search tree pruning with Branch-and-Bound, while the



(a) 2 Processors (b) 8 Processors

Figure 9. Performance comparison of algorithm running times (LBBD vs. Integrated).



(a) 2 Processors (b) 8 Processors

Figure 10. Throughput improvement compared to the heuristic mapping and scheduling algorithm in [4].

`LBBD` approach only invokes the theory solver on full solutions after all mapping/ordering decisions are made. However, `LBBD` is a more general framework that has broader applicability with different master and sub-problem solvers than the `Integrated` approach, which is limited to integrating a SAT solver with a theory solver.

Fig. 10 shows the improvement in throughput of the SAT approach compared to the heuristic mapping and scheduling algorithm in [4]. Both `LBBD` or `Integrated` return the same throughput value with different running times. We use the throughput value obtained with the SAT approach as the baseline, plotted as the line with value 1, and plot the throughput obtained with the algorithm in [4] relative to the baseline. Results show that the SAT approach can achieve significant improvement in throughput compared to the heuristic algorithm.

7 Related Work

Some authors have explored integration of SAT with application-specific theory solvers. Metzner *et al* [17] combined a SAT solver with real-time scheduling theory to address the problem of periodic task allocation to achieve schedulability. Satish *et al* [14] used a SAT solver to encode both task mapping and scheduling, and graph theoretic techniques to detect cycles and critical paths. Both [17] and [14] take task graphs as input, and only consider non-overlapped schedules with the objective of minimizing the overall schedule length (makespan). Task graphs can be viewed a special case of acyclic HSDF graphs without any initial tokens, since any cycle indicates a deadlock, while cycles are allowed in a HSDF graph as long as there are non-zero number of tokens on the cycle, hence actor firing does not always have to follow edge precedence order as in task graphs.

Stuijk *et al* [4] presented a design flow of mapping multiple applications (each application is a SDF graph) onto a multiprocessor platform with the objective of minimizing resource usage while meeting a given throughput constraint. TDMA scheduling is used to achieve performance isolation among applications, and static order scheduling is used for actors within the same application. The optimization goal is to minimize resource usage (TDMA wheel size allocated to an application) under a given throughput constraint, thus maximizing the number of applications that can run concurrently on the system while providing throughput guarantees. The authors used a heuristic weighted cost function for actor-to-processor mapping that combines processing load, memory load, communication and latency load; a list scheduler for static order scheduling, and binary search for TDMA time slice allocation. Moreira *et al* [10] addressed a similar problem as [4], i.e., mapping a SDF graph onto a multiprocessor platform with a combination of TDMA scheduling and static order scheduling to meet throughput and latency constraints while

minimizing resource usage. They used exhaustive search on top of a list scheduler for static order scheduling, and presented a linear programming formulation for assigning TDMA time wheel sizes by distributing timing slack among actors in the SDF graph to maximize the sum of deadline extensions. Compared to our work, [4] and [10] addressed the more general and difficult design problem of mapping multiple applications onto a heterogeneous hardware platform, while we address the problem of mapping a single application onto a homogeneous hardware platform. The techniques in this paper can be integrated into the overall design flow in [4] or [10] as refinements or replacements of a few key steps, and are compatible and complementary to the other design steps in their design flows that address mapping of multiple applications. It is not difficult to extend the SAT-based techniques to handle a heterogeneous platform, where actor execution times may be different on different processors, by adding additional encoding variables.

Stuijk *et al* [3] presented an efficient state space exploration technique for calculating the Pareto tradeoff curve between buffer size and the maximum achievable throughput of a SDF graph without resource constraints. It works by firing all enabled actors whose dependency constraints are satisfied in a self-timed manner, assuming there is an unlimited number of available processors, and detecting a cycle in the state space. This is a polynomial-time deterministic algorithm that is guaranteed to either terminate or result in a deadlock. However, when the number of processors is limited and is less than the maximum number of concurrently enabled actors, we need to make a scheduling decision to choose a subset of actors to fire among all enabled actors at each step when some actor finishes firing. This makes the problem NP-complete, since there are exponentially many search tree branches to be explored in order to find the optimal schedule with maximum throughput. A heuristic list scheduler is used in [4] to construct static order schedules, but it may be important for certain cost-sensitive and resource-constrained embedded applications to search all possible static order schedules to find the optimal solution. One approach is to use an outer loop to search all possible static order schedules exhaustively, invoking the throughput computation algorithm as a subroutine for each schedule, as done in [10]. In this paper, we aim to push the scalability limit in the quest for optimal solutions using modern SAT solvers with the hope to achieve better scalability than explicit exhaustive enumeration, since SAT solvers have matured and shown great advancement in scalability over the past years, and are capable of solving many NP-complete problems with reasonable size and complexity,

Govindarajan *et al* [18] presented an ILP formulation of resource-constrained rate-optimal software pipelining, equivalent to overlapped scheduling of HSDF graphs with limited number of processors, assuming that actor-to-processor mapping is already given. An ILP problem is constructed for each possible iteration period. The variables are actor execution start times, and the linear constraints encode mutual exclusion of processor resources. If there is no solution, i.e., the scheduling problem is infeasible for the given iteration period, then increase the iteration period, and vice versa, until the smallest iteration period that permits a feasible schedule is found. This can be done with binary search. This approach is not very efficient due to the binary search for the iteration period.

Govindarajan *et al* [19] presented a LP formulation of optimal static scheduling of a multirate SDF graph with minimum buffer size distribution while guaranteeing the maximum achievable throughput without resource constraints (assuming unlimited number of processors), by assuming a known iteration period obtained by converting the SDF graph into HSDF graph and performing MCR calculation. The LP variables are actor execution start times, and the linear constraints encode the fact that any edge buffer neither underflows or overflows. Wiggers *et al* [8] presented a conservative algorithm for obtaining the minimum buffer size distribution for a SDF graph to meet a given throughput guarantee, with the assumption that the underlying OS scheduling algorithm supports resource guarantees, e.g., resource reservation or TDMA scheduling, so that each actor's Worst-Case Response Time (WCRT) can be statically determined. Compared to [19] and [3], their algorithm is very efficient but may yield conservative and potentially inaccurate results. The algorithms in [19], [8] and [3] can be used to obtain an initial buffer size distribution in our approach.

In this paper, we have assumed a hard real-time network-on-chip hardware platform where inter-processor communication has constant delay and bandwidth independent of the application workload, which may be accurate for hardware platforms such as network-on-chip with high communication bandwidth compared to application demands. However, the core contribution of this paper does not depend on the simplistic network model, which can be easily replaced by more sophisticated network models, e.g., Moonen *et al* [20] presented a SDF model for a network-on-chip in which a hardware arbiter is used to decouple communication from computation; Poplavko *et al* [21] presented a HSDF model based on IPC graphs [1] for a network-on-chip system.

For *bus-based* hardware platforms instead of network-on-chip, network communication delay depends on the bus arbitration policy and application workload. Sriram *et al* [1] modeled the self-timed implementation of a HSDF graph on a bus-based multiprocessor platform with a IPC graph, constructed by adding communication actors to model data send and receive. If communication and computation are self-timed, then it is not possible to describe the application as a static IPC graph with bus contention, since the execution order among communication actors is not fixed. Khandelia *et al* [22] showed that it can be advantageous to impose a static total order on all actors modeling inter-processor communication transactions (send and receive) for a shared-bus based system, since it obviates the need to model packet contention on the bus and enables using traditional MCR analysis to calculate throughput. Furthermore, ordered transaction schedules can often (although not always) outperform self-timed schedules in terms of throughput. If we consider this bus-based architecture, then we need to add another dimension to the search space, a *global total order* of all communication actors on all processors. We have in fact added it to the the SAT encoding discussed in Section 5, but omit it from this paper since it does not bring much new technical insight beyond our current model.

8 Conclusions

In this paper, we have presented SAT-based techniques for addressing the problem of mapping HSDF graphs onto a multiprocessor platform with the objective of maximizing application throughput, and showed that they are effective and can scale up to applications with reasonable size and complexity.

The techniques presented in this paper are only applicable to HSDF graphs, not to multirate SDF graphs, since the MCR technique for calculating throughput is not directly applicable to multirate SDF graphs. Any SDF graph can be transformed to its equivalent HSDF graph with a potentially large increase in size. Our experience shows that the MCR computation algorithm becomes the computational bottleneck that prevents direct application of the our approach to the converted HSDF graph. The technique in [3] can be used to obtain the maximum throughput of a SDF graph without conversion to HSDF using cycle detection in the state space. However, in contrast to a critical cycle in the HSDF graph, the state space cycle does not have a direct correspondence to a cycle in the SDF graph. Therefore, we cannot achieve effective search space pruning by adding blocking clauses in SAT, and must resort to exhaustive enumeration or heuristic mapping and scheduling algorithms, e.g., the one in [4]. *Actor or task clustering* can be used to alleviate the problem of exponential size increase when transforming a SDF graph to a HSDF graph, where a group of actors is mapped and scheduled onto a multiprocessor platform as a single unit. Pino *et al* [23] presents some heuristic actor clustering techniques to reduce the number of actors before the transformation. Kiazad *et al* [24] evaluates a number of clustering and cluster scheduling algorithms when used in the context of evolutionary algorithms for task graph mapping and scheduling on multiprocessors. We view the body of work on actor/task clustering as complementary to our techniques, since each actor can also be viewed as a cluster of actors without affecting the rest of the algorithm.

Acknowledgements

This work was partially supported by Hong Kong RGC CERG #613506, NSERC Discovery Grant #341823-07, and NSFC Grant #60736017. We would like to thank Sander Stuijk and Orlando Moreira for helpful discussions, and Niklas Sörensson for his help on Minisat.

References

[1] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, 2000.

[2] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Computers*, vol. 40, no. 2, pp. 178–195, 1991.

[3] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC*, E. Sentovich, Ed. ACM, 2006, pp. 899–904.

[4] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *DAC*, 2007, pp. 777–782.

[5] S. Stuijk, "Predictable mapping of streaming applications on multiprocessors," Ph.D. dissertation, Technical University of Eindhoven, 2007.

[6] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 889–899, 1998.

[7] Model-Driven Architecture (MDA), url=http://www.omg.org.

[8] M. Wiggers, M. Bekooij, P. G. Jansen, and G. J. M. Smit, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *CODES+ISSS*, R. A. Bergamaschi and K. Choi, Eds. ACM, 2006, pp. 10–15.

[9] S. Sriram and E. A. Lee, "Determining the order of processor transactions in statically scheduled multiprocessors," *VLSI Signal Processing*, vol. 15, no. 3, pp. 207–220, 1997.

[10] O. Moreira, F. Valente, and M. Bekooij, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *EMSOFT*, C. M. Kirsch and R. Wilhelm, Eds. ACM, 2007, pp. 57–66.

[11] F. Heras, J. Larrosa, and A. Oliveras, "MiniMaxSat: An efficient Weighted Max-SAT solver," *Journal of Artificial Intelligence Research*, 2008.

[12] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[13] J. N. Hooker and H. Yan, "Logic circuit verification by benders decomposition," *Principles and Practice of Constraint Programming: The Newport Papers*, p. 267C288, 1995.

[14] N. Satish, K. Ravindran, and K. Keutzer, "A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors." in *DATE*, R. Lauwereins and J. Madsen, Eds. ACM, 2007, pp. 57–62.

[15] H. M. Sheini and K. A. Sakallah, "From propositional satisfiability to satisfiability modulo theories," in *SAT*, ser. Lecture Notes in Computer Science, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 1–9.

[16] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA,

USA, June 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3

[17] A. Metzner and C. Herde, "RTSAT– An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures." in *RTSS*. IEEE Computer Society, 2006, pp. 147–158.

[18] R. Govindarajan, E. R. Altman, and G. R. Gao, "A framework for resource-constrained rate-optimal software pipelining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 11, pp. 1133–1149, 1996.

[19] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *VLSI Signal Processing*, vol. 31, no. 3, pp. 207–229, 2002.

[20] M. B. Arno Moonen and J. van Meerbergen, "Timing analysis model for network based multiprocessor systems," in *Proc. PROGRESS Symposium on Embedded Systems*, 2004.

[21] P. Poplavko, T. Basten, M. Bekooij, J. L. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *CASES*, J. H. Moreno, P. K. Murthy, T. M. Conte, and P. Faraboschi, Eds. ACM, 2003, pp. 63–72.

[22] M. Khandelia, N. K. Bambha, and S. S. Bhattacharyya, "Contention-conscious transaction ordering in multiprocessor dsp systems," *IEEE Transactions on Signal Processing*, vol. 54, no. 2, pp. 556–569, 2006.

[23] J. L. Pino and E. A. Lee, "Hierarchical static scheduling of dataflow graphs onto multiple processors," in *ICASSP*, 1995, pp. 2643–2646.

[24] V. Kianzad and S. S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 7, pp. 667–680, 2006.

Appendix I

Here is the proof of Theorem 1 in Section 4.

Proof. If we adopt the buffer allocation in Equation (2), then the MCR of the HSDF graph after buffer allocation is:

$$\text{MCR}_1 = \max(\frac{t_{in}}{\lceil B_{ij} * t_{in}/t_{ijnn} \rceil}, \frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor})$$

Since

$$\frac{t_{in}}{\lceil B_{ij} * t_{in}/t_{ijnn} \rceil} \leq \frac{t_{in}}{B_{ij} * t_{in}/t_{ijnn}} = \frac{t_{ijnn}}{B_{ij}}$$

and

$$\frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor} \geq \frac{t_{jn}}{B_{ij} * t_{jn}/t_{ijnn}} = \frac{t_{ijnn}}{B_{ij}}$$

We have

$$\text{MCR}_1 = \frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor} \tag{12}$$

Similarly, if we adopt the buffer allocation in Equation (4), then the MCR of the platform-enhanced HSDF graph is:

$$\begin{aligned} \text{MCR}_2 &= \max(\frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor}, \frac{t_{jn}}{\lceil B_{ij} * t_{jn}/t_{ijnn} \rceil}) \\ &= \frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor} \end{aligned} \tag{13}$$

In order to minimize the MCR of the platform-enhanced HSDF, we need to choose between the two buffer allocation schemes based on the values of MCR_1 and MCR_2, which explains the two *if* conditions (3) and (5) in the theorem. The final MCR is:

$$\text{MCR}_{min} = \min(\text{MCR}_1, \text{MCR}_2) \tag{14}$$

Next, we show that any other buffer allocation scheme will result in a MCR that is larger than MCR_{min}. Without loss of generality, assume Condition (3) is true, and Equation (2) is used as the buffer allocation scheme. Consider another buffer allocation:

$$\text{buf}'_{in} + \text{buf}'_{in} = B_{ij}$$

where $\text{buf}'_{in} \neq B_{in}$ and $\text{buf}'_{in} \neq B_{in}$. Using a positive integer $x \geq 1$ to denote the difference between them, either

$$\text{buf}'_{in} = B_{in} + x, \text{buf}'_{jn} = B_{jn} - x \tag{15}$$

or

$$\text{buf}'_{in} = B_{in} - x, \text{buf}'_{jn} = B_{jn} + x \tag{16}$$

If Equation (15) holds, then the resulting MCR is:

$$\begin{aligned} \text{MCR}'_1 &= \max(\frac{t_{in}}{\lceil B_{ij} * t_{in}/t_{ijnn} \rceil + x}, \frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor - x}) \\ &\geq \frac{t_{jn}}{\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor - x} \\ &\geq \frac{t_{jn}}{(\lfloor B_{ij} * t_{jn}/t_{ijnn} \rfloor)} \\ &= \text{MCR}_1 \end{aligned} \tag{17}$$

If Equation (16) holds, then the resulting MCR is:

$$\begin{aligned} \text{MCR}'_1 &= \max(\frac{t_{in}}{(\lceil B_{ij} * t_{in}/t_{ijnn} \rceil - x)}, \frac{t_{jn}}{\lfloor \text{buf}'_{ij} * t_{jn}/t_{ijnn} \rfloor + x}) \\ &\geq \frac{t_{in}}{\lceil B_{ij} * t_{in}/t_{ijnn} \rceil - x} \\ &\geq \frac{t_{in}}{\lfloor B_{ij} * t_{in}/t_{ijnn} \rfloor} \\ &= \text{MCR}_2 \end{aligned} \tag{18}$$

From Equations (17) and (18), we know that $\text{MCR}'_1 \geq \min(\text{MCR}_1, \text{MCR}_2) = \text{MCR}_{min}$, that is, any other buffer allocation that is different from Equation (2) if Condition (3) holds will result in a higher MCR value than the buffer allocation in Equation (2). We can reach a similar conclusion for the case when Condition (5) holds. Therefore, Theorem 1 is proven. □